Journal of Computer Applications
计算机应用,2013,33(8):2375 -2378
ISSN 1001-9081
CODEN JYIIDU
2013-08-01
http://www.joca.cn

文章编号:1001-9081(2013)08-2375-04
doi:10.11772/j.issn.1001-9081.2013.08.2375

基于 Trie 树的相似字符串查找算法

刘丽霞[1,2*],张志强[2]
(1.闽南理工学院 信息管理系,福建 石狮 362700; 2.哈尔滨工程大学 计算机科学与技术学院,哈尔滨 150001)
(*通信作者电子邮箱 llx0454@126.com)

摘 要:基于 Trie 树的相似字符串查找算法是利用编辑距离的阈值来计算每个节点的活跃节点集,已有算法由于存在大量的冗余计算,导致时间复杂度和空间复杂度都比较高。针对这个问题,采用了基于活跃节点的对称性和动态规划算法的思想对已有算法进行改进,并对活跃节点集进行了修剪,提出了 New-Trie-Stack 算法。该算法避免了活跃节点的重复计算,以及已有算法在保存所有已遍历节点的活跃节点集时的空间开销。实验结果表明 New-Trie-Stack 算法在时间复杂度和空间复杂度上都有明显的下降。

关键词:Trie 树;相似字符串;编辑距离;活跃节点;动态规划
中图分类号:TP391.3 **文献标志码**:A

Similar string search algorithm based on Trie tree

LIU Lixia[1,2*], ZHANG Zhiqiang[2]
(1. *Department of Information and Management, Minnan University of Science and Technology, Shishi Fujian* 362700, *China*;
2. *College of Computer Science and Technology, Harbin Engineering University, Harbin Heilongjiang* 150001, *China*)

Abstract: Similar string search algorithms based on Trie tree need to compute active-node set of a node by editing distance threshold. A large number of redundant computation leads to a high time and space complexity. A new algorithm named New-Trie-Stack was proposed, which utilized the symmetrical properties of active-node set and the dynamic programming method to improve the performance. It could avoid the redundancy cost on active-node set computing and storing; moreover, active-node sets were pruned. The experimental results show that New-Trie-Stack algorithm has lower time complexity and space complexity.

Key words: Trie tree; similar string; edit distance; active-node; dynamic programming

《计算机应用》Journal of Computer Applications

0 引言

在许多应用中,相似性连接是一种基本的操作,如 DNA 分析、拼字检查、语音辨识、抄袭侦测等。而现有的相似性连接算法中通常是利用过滤加验证的框架来处理字符串之间的相似性。过滤阶段:为每一个字符串生成一个标签,利用标签产生候选字符串对;验证阶段:验证候选字符串对,并输出最终结果,如 Pass-Join、Ed-Join、Trie-Join 等[1-10]。然而这些算法都存在着一些缺点:1)不利于处理短字符串的数据集(平均字符串的长度不超过 30 个字符);2)需要建立大量的字符串索引;3)需要耗费大量成本来支持数据集的动态更新[5]。

为了提高现有算法的性能,本文对文献[5]的 Trie-Join 算法进行了改进,利用编辑距离的限制并采用动态规划的方式遍历树节点,从而高效地查找出相似字符串。

1 相关概念

1.1 Tire 树

Trie 树即单词查找树,经常被搜索引擎系统用于文本词频统计。由于很多的字符串都共享相同的字符串前缀,而 Trie 树的优点就是能最大限度地减少字符串重复前缀的无谓比较。本文所建立的 Trie 树中,根节点不保存字符,每个节点都有一个索引号唯一标识自己,如图 1 所示。

1.2 编辑距离

编辑距离是指一个字符串经过有限次编辑操作(插入、删除、修改)变为另一个字符串,设字符串 $Str1$ 转换成字符串 $Str2$ 的编辑距离记为 $ED(Str1, Str2)$。设 R 为字符串集合,一种简单的查找相似字符串的方法是列出全部字符串对$\langle r, s\rangle \in R$,逐一计算它们的编辑距离。然而,这种解决办法存在大量的冗余计算。事实上,在大多数情况下,我们不需要计算两个字符串之间完整的编辑距离,利用动态规划算法可以提前终止,参见文献[5]。

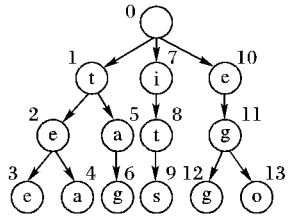


图 1 字符串集合的 Trie 树例子

1.3 活跃节点

节点之间的编辑距离即为节点所对应的字符串之间的编辑距离。Trie 树上某一节点 n 的活跃节点是指这样的一些节点,它们到节点 n 所需的编辑距离不大于阈值 τ。

2 基于 Trie 树的相关算法介绍

2.1 Trie-Node 算法

Trie-Node 算法采用了 Trie 树来存储字符串集合,从根节点到某一叶子节点的路径唯一对应一个字符串。前序遍历

收稿日期:2013-03-05;**修回日期**:2013-05-03。 **基金项目**:国家自然科学基金资助项目(61272184, 61202090, 61100007)。
作者简介:刘丽霞(1986 -),女,黑龙江桦南人,助教,硕士,主要研究方向:信息检索; 张志强(1973 -),男,河北雄县人,教授,博士,CCF 高级会员,主要研究方向:信息检索、智能信息处理、数据库。

Trie 树节点,利用动态规划的方式把计算字符串之间的编辑距离改成计算节点之间的编辑距离,因为 Trie 树的每一个节点都对应着一个字符串或字符串的前缀。对于树上的任一节点 n,将编辑距离小于阈值 τ 的节点作为 n 的活跃节点。通过逐一计算树上节点的活跃节点集来找出全部的相似字符串对。每一个节点的活跃节点集都是利用它的父节点的活跃节点集计算得来的。当遍历到某一叶子节点 n 时,如果叶子节点 m 是节点 n 的活跃节点,那么〈n, m〉就组成了一对相似节点,它们所对应的字符串就是一对相似字符串[5-6,11-12]。

Trie-Node 算法避免了字符串之间相同前缀的重复计算,大大减少了运行时间。然而,在计算每一个节点的活跃节点集时,我们发现活跃节点具有着对称性,如果节点 n 是节点 m 的活跃节点,那么节点 m 同样也是节点 n 的活跃节点,所以 Trie-Node 算法仍然存在很多重复计算。

2.2 Trie-DPStack 算法

根据活跃节点的对称性,Trie-DPStack 算法动态地计算每一个节点的活跃节点集,即只与已经遍历过的节点进行编辑距离的计算,并将其加入到与它互为活跃节点的活跃节点集中;同时利用一个栈来存储遍历过的节点和节点的活跃节点集,栈的大小为树的最大深度。

Trie-DPStack 算法的具体思想是首先将字符串集合构建成一棵 Trie 树,创建一个栈,前序遍历树中的节点,通过父节点的活跃节点集计算当前遍历节点的活跃节点集,同时修改栈中与它的编辑距离小于阈值的节点的活跃节点集,然后将遍历过的节点及其活跃节点集入栈。如果当前节点没有孩子节点待遍历时,退栈,这样保证了每个节点在进栈前栈顶元素为它的父节点。当遇到叶子节点时,则输出该节点的相似字符串对,退栈,遍历其他节点,直到栈为空,输出所有的相似字符串对。

Trie-DPStack 算法避免了 Trie-Node 算法中节点之间的重复计算,同时引入栈也避免了保存索引记录时所增加的存储空间开销[5]。然而在计算活跃节点集时,可以根据父节点的活跃节点与当前遍历节点的关系,继续对算法进行优化。

3 基于 Tire 树的改进算法描述

为进一步提高算法性能,本文优化了 Trie-DPStack 算法,定义改进算法为 New-Trie-Stack。通过分析发现无论采用哪种算法,每个节点的活跃节点集都是由父节点的活跃节点集计算得来的,而父节点的活跃节点集中一般包括:父节点、祖先节点、当前节点的兄弟节点和其他分支上的节点。假设阈值 $\tau = 1$ 时,父节点和兄弟节点到当前节点的编辑距离均为 1,祖先节点到当前节点的编辑距离至少为 2,其他分支上的节点到当前节点父节点的编辑距离为 1。进一步观察还发现,由于父节点或祖先节点所对应的只能是字符串的前缀,所以不可能作为相似字符串输出。据此,计算活跃节点集时可以直接将父节点和祖先节点删除掉。为每个节点附加一个索引标号,以前序遍历的顺序为每个节点编号[13]。

3.1 计算活跃节点集

利用栈来暂时存储访问过的树节点及该节点的活跃节点集,没有孩子节点可遍历的要退栈,保证每个节点入栈前它的父节点为栈顶元素,这样栈中保存的一直都是 Trie 树的一个分支,也就是栈中的节点是当前遍历节点的父节点和祖先节点。每个节点以结构体的形式存储,包括节点的索引号、孩子节点数组等,这样就可以体现出节点之间的父子关系和兄弟关系。例如图 2 中当节点 3 入栈时,栈顶节点 2 是 3 的父节点,而栈中其他节点都是节点 3 的祖先节点,节点 2 的孩子节点即为节点 3 的兄弟节点。

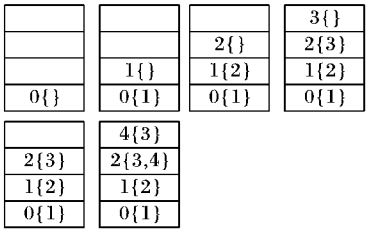


图 2 利用栈来遍历 Trie 树节点查找相似字符串对的例子

根据编辑阈值的限制来查看父节点的活跃节点,有以下四种情况:

1)如果节点为父节点,则只需将当前遍历节点加入到父节点的活跃节点集中。

如图 1 中节点 1 是节点 2 的父节点,设 $\tau = 1$。节点 1 是节点 2 的活跃节点,节点 1 所对应的字符串"t"是节点 2 对应字符串"te"的前缀,所以它们不可能作为相似字符串对输出,因此可以得证父节点不能与它的子节点或子孙节点作为一对相似字符串,而且它也不会影响子孙节点的活跃节点集计算。

2)如果节点为祖先节点,并且编辑距离不大于阈值 τ,将当前节点加入到祖先节点的活跃节点集中。

设 $\tau > 1$,如图 1 假设前序遍历到图中节点 3 时,通过父节点 2 的活跃节点集来计算节点 3 的活跃节点集,如图 2 所示,节点 3 的祖先节点 0 和 1 的其他子孙节点的活跃节点集是通过 0 和 1 的活跃节点集计算获得的,因此如果节点 3 与祖先节点的编辑距离不大于阈值,那么它与祖先节点的其他子孙节点也可能互为活跃节点,如图 1 中的节点 4 和 5。所以得证子孙节点的活跃节点是间接地通过祖先节点的活跃节点集计算得来的。

3)如果节点为兄弟节点,直接将兄弟节点加入到当前节点的活跃节点集中。

因为父节点的活跃节点都是已经遍历过的节点,所以兄弟节点及它的子孙节点也是已经遍历过的节点,而兄弟节点之间拥有着相同的字符串前缀,它们的编辑距离为 1,如图 1 的节点 3 和 4,于是得证如果节点为兄弟节点,那么它们互为活跃节点。

4)如果节点为其他分支上的节点,则计算该节点及其子孙节点与当前遍历节点的编辑距离,将编辑距离不大于阈值 τ 的节点加入到当前节点的活跃节点集中。

假设两个分支上的节点 m 和 n 互为活跃节点,编辑距离为 τ,节点 s 是 n 的孩子节点。节点 s 与节点 m 的编辑距离为 $\tau + \theta$(节点 s 和 m 不同时,θ 为 1,否则为 0),如果节点 s 与节点 m 不同,而节点 s 的某个孩子节点 t 与节点 m 相同,则节点 t 与节点 m 的编辑距离仍然为 τ。于是得证,其他分支上的节点与当前遍历节点的编辑距离大于 τ 时,它的子孙节点与当前遍历节点的编辑距离可能等于 τ。

下面是遍历树节点的例子:假设阈值 $\tau = 1$ 时,先序遍历图 1 中的 Trie 树,首先根节点 0 入栈,然后先序遍历节点 1,将节点 1 加入到父节点 0 的活跃节点集中,同时根据它的父节点的活跃节点集计算节点 1 的活跃节点,如果计算出的活跃节点也在栈中,要将节点 1 加入到这些节点的活跃节点集中;接下来访问节点 1 的第一个孩子节点 2,将 2 加入到 1 的活跃节点集;通过节点 1 计算节点 2 的活跃节点,继续访问节点 3,通过 2 计算 3 的活跃节点集,同时将 3 加入到栈中它的活跃节点的活跃节点集中,由于 3 是叶子节点,所以查看 3 的

《计算机应用》Journal of Computer Applications

活跃节点集中是否有叶子节点,3 退栈;接下来访问节点 4,根据 2 计算 4 的活跃节点集,2 的活跃节点 3 与 4 是兄弟节点,将 3 加入到节点 4 的活跃节点集中,2 是 4 的父节点,且编辑距离正好为 1,不需将 2 加入到节点 4 的活跃节点集中,同时将 4 加入到栈中它的活跃节点的活跃节点集中,由于 4 同样是叶子节点,所以查看 4 的活跃节点集中是否有叶子节点,由于叶子节点 3 是 4 的活跃节点,则〈4,3〉作为相似字符串输出。继续遍历其他节点,直到栈为空,则所有的相似字符串全部被输出。节点进栈的过程如图 2 所示。

3.2　计算节点之间的编辑距离

采用动态规划的算法来计算字符串之间的编辑距离,例如求字符串 $r = r_1r_2\cdots r_n$ 和 $s = s_1s_2\cdots s_m$ 的编辑距离,定义 $ED(i,j)$ 为前缀 $r_1r_2\cdots r_i$ 和前缀 $s_1s_2\cdots s_j$ 的编辑距离。

$$ED(i,j) = \begin{cases} ED(i-1,j-1), & r_i = s_j \\ \min(ED(i-1,j-1), ED(i-1,j), \\ \quad ED(i,j-1)) + 1, & r_i \neq s_j \end{cases} \quad (2)$$

设 $\boldsymbol{ED}$ 为 $n+1$ 行和 $m+1$ 列的矩阵,$ED(0,j) = j$ $(0 \leqslant j \leqslant n)$,计算矩阵 $\boldsymbol{ED}$,由编辑距离阈值 τ 的限制可以提前结束程序。

3.3　New-Trie-Stack 算法描述

输入:字符串集合 R,编辑距离阈值 τ;

输出:相似字符串集合 P。

```
T = CreateTRIE (R);                          //建立 Trie 树
S = stack (h);                    // 建立栈 S,h 为 Trie 树深度
Pre_Order (TRIENode)
          //前序遍历 Trie 树节点,计算活跃节点集并入栈。
while (TRIENode)                          //节点 TRIENode 存在
{ Compute_ActiveNode (TRIENode)
                        //计算节点 TRIENode 的活跃节点集
  { Estimate(Parent_ActiveSet)
  //通过父节点的活跃节点集计算节点 TRIENode 的活跃节点集
  if (Parent_Active [i] = = forefather)     //如果节点为父节点
    ComputeDistance (TRIENode, Parent_Active [i]) < τ,
      修改 Parent_Active [i] 的活跃节点集
  if (Parent_Active [i] = = brother)        //如果节点为兄弟节点
    将Parent_Active [i] 加入到 TRIENode 的活跃节点集
    ComputeDistance (TRIENode,
      Parent_Active [i] _de-scendants) < τ
    将Parent_Active [i] _descendants 加入到 TRIENode 的
      活跃节点集
  else                              //如果节点为其他分支上的节点
    ComputeDistance (TRIENode, Parent_Active [i]) < τ
    ComputeDistance (TRIENode,
      Parent_Active [i] _de-scendants) < τ
    将Parent_Active [i] 或 Parent_Active [i] _descenda-nts 加入到
      TRIENode 的活跃节点集}
  TRIENode, ActiveNode (TRIENode) 入栈;
          //节点 TRIENode 和 TRIENode 的活跃节点集入栈
  IsLeaf (TRIENode)                         //判断是否是叶子节点
  Output(相似字符串对);
  //将活跃节点集中与 TRIENode 相似的
  //节点作为相似节点对输出
  TRIENode 退栈;}
//如果节点 TRIENode 是叶子节点或
//全部子节点都已经遍历过,退栈
```

4　实验结果对比与分析

本文实验采用的硬件环境为 Intel Pentium 2.93 GHz CPU、4 GB内存计算机,操作系统为 Windows XP,算法代码采用C++实现。

4.1　在 English Vocabulary 上的实验

本数据集为英语词汇表,共 4 795 个词汇。

4.1.1　字符串的数量对算法运行时间的影响

通过改变单词的数量来观察三种算法的运行时间,结果如图 3 所示。可以看出三种算法的运行时间都随着字符串的增加而增加,用时最多的是 Trie-Node 算法,而 Trie-DPStack 和 New-Trie-Stack 的时间增长率要小于 Trie-Node,因为节点数量的增多使得 Trie-Node 算法的活跃节点的重复计算量加大,而 Trie-DPStack 和 New-Trie-Stack 利用了活跃节点的对称性,使得计算量减少了近一半。New-Trie-Stack 算法在计算活跃节点上对 Trie-DPStack 算法进行了改进,分类计算父节点的活跃节点集,减少了很多不必要的计算,因此运行时间比 Trie-DPStack 更短。同时还发现当字符串增加时,它们之间的时间差越大。

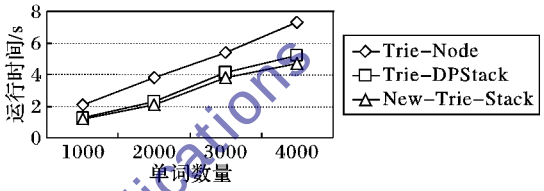


图 3　不同字符串数量时各算法的运行时间对比

4.1.2　Trie 树节点的数量对算法运行时间的影响

抽取数据集 English Vocabulary 以“a”开头的单词,共计 102 个,Trie 树节点 503 个。通过限制节点的个数,来对比三种算法的运行时间。表 1 给出了当改变 m 的值时,各种算法的运行时间,m 表示 Trie 树的节点数(索引号为 1 ~ m)。

表 1　限制节点数量时各算法的运行时间对比　　s

算法	m			
	100	200	300	400
Trie-Node	0.106	0.664	1.147	1.937
Trie-DPStack	0.054	0.337	0.613	1.138
New-Trie-Stack	0.048	0.318	0.569	1.024

通过分析表 1 发现,Trie 树的节点数量也是影响算法运行时间的因素之一,相同数量的字符串构建的 Trie 树节点数量不一定相同,因此对于字符串集中的字符串来说,如果相同前缀的字符串数量比较多时,利用 Trie 树查找相似字符串的效率会更明显。不难观察 New-Trie-Stack 算法的运行时间最少。

4.1.3　字符串的平均长度对算法运行时间的影响

通过改变字符串集合中字符串的平均长度,来进一步分析三种算法的运行时间。图 4 分别给出了当阈值为 1 和 2 时,各算法的运行时间随着字符串平均长度变化而变化的对比。可以看出,当字符串的平均长度变大时,各算法的运行时间都在增加,然而变化比率在逐渐减小,当字符串的平均长度为 40 ~ 50 时,三种算法的运行时间均趋于稳定;而 New-Trie-Stack 算法同样适用于处理长字符串集合,并且在运行效率上明显优于 Trie-Node 和 Trie-DPStack。

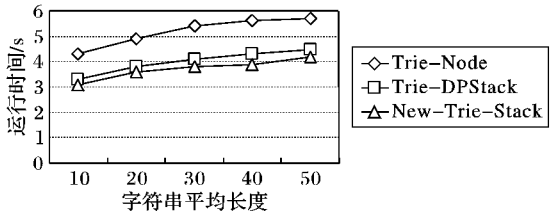
4.2　在 AOL Query Log 上的实验

AOL Query Log 为美国在线查询日志[5],共有 2 000 条查询。同样利用三种算法分别在 AOL Query Log 数据集上进行实验,通过改变阈值的方式,观察三种算法的运行时间,结果如图 5 所示。

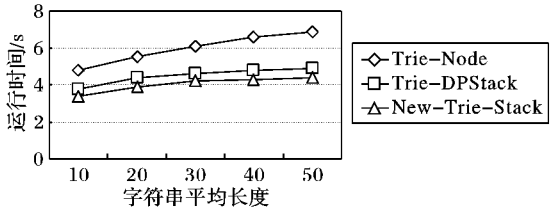
阈值也同样影响着算法的运行时间。当阈值增大时,每一个节点的候选活跃节点数增多,也就是说对于父节点的每

Journal of Computer Applications 《计算机应用》

一个活跃节点来说，还需要继续遍历它的子孙节点，找出编辑距离小于阈值的节点，因此增加了计算量；然而采用动态规划的方式遍历树节点，使得 Trie-DPStack 和 New-Trie-Stack 在时间消耗上明显低于 Trie-Node。New-Trie-Stack 算法在计算活跃节点时，缩减了计算量，进一步地提高了算法的运行效率。



(a) 编辑距离阈值=1



(b) 编辑距离阈值=2

图 4 字符串平均长度对各算法运行时间的影响对比

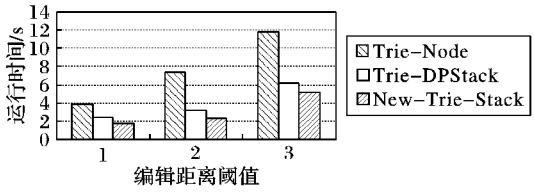


图 5 不同阈值时各算法的运行时间对比

4.3 算法的复杂度分析

Trie-Node 算法在最坏的情况下，当前遍历节点 m 的时间复杂度为 $O(\tau \cdot |A_m|)$，$|A_m|$ 为节点 m 的活跃节点的数量，因为每一个活跃节点只能通过祖先节点的活跃节点集经过 τ 步计算得来，所以 Trie-Node 算法的时间复杂度为 $O(\tau \cdot |A_T|)$，A_T 是 Trie 树上所有的活跃节点的数量。当遍历某一节点时，需要保存它的祖先节点的活跃节点，对于一个叶子节点 n，令 $C(n)$ 为节点 n 的祖先节点的活跃节点的数量，C_{max} 为 Trie 树上所有叶子节点中 $C(n)$ 最大的，所以空间复杂度为 $O(|T| + C_{max})$，$|T|$ 是 Trie 树节点的数量[5]。

Trie-DPStack 算法利用活跃节点的对称性，减少了 Trie-Node 算法的计算量，时间复杂度为 $O(\tau/2 \cdot |A_m|)$；同时它采用动态规划的方式，用一个栈来存储遍历的节点，所以空间复杂度为 Trie 树的最大深度 h。

New-Trie-Stack 算法在 Trie-DPStack 算法的基础上对节点的活跃节点集进行了修剪，祖先节点不需要加入到子孙节点的活跃节点集；而且我们发现祖先节点如果和它的某一子孙节点的编辑距离为阈值 τ 的最大值时，那么子孙节点的子孙节点就不再影响祖先节点的活跃节点集了，这样如果节点与祖先节点的步长为 τ，则不需要计算它们之间的编辑距离（因为祖先节点是子孙节点的前缀节点），所以时间复杂度为 $O(\tau/2 \cdot |A_m|) - (|T| - |T_\tau|)$，$|T_\tau|$ 为节点深度小于 τ 的节点数量，因为这些节点没有步长为 τ 的祖先；空间复杂度仍然为 Trie 树的最大深度 h。

通过三种算法的复杂度分析可以发现，影响算法复杂度的主要因素是 Trie 树中活跃节点的总数量，New-Trie-Stack 算法优化了 Trie-Node 算法和 Trie-DPStack 算法，进一步地优化了 Trie 树的构造，修剪了每个节点的活跃节点集，提高了算法的效率[14-15]。

5 结语

本文通过编辑距离阈值的限制来研究字符串的相似性查找算法，提出了基于 Trie 树的字符串相似性连接算法的改进算法 New-Trie-Stack。该算法利用 Trie 结构索引字符串，依据对已有算法 Trie-DPStack 的分析[5]，对节点的活跃节点集的计算方法进行优化，同时改进了节点之间的编辑距离计算。实验验证该算法在 Trie-DPStack 算法的基础上进一步提高了算法效率，能够更快速地查找出全部的相似字符串。

参考文献：

[1] LI G L, DENG D, WANG J N, *et al*. Pass-Join: a partition-based method for similarity joins [J]. Proceedings of the VLDB Endowment, 2011, 5(3): 253 - 264.

[2] JESTES J., LI F F, YAN Z P, *et al*. Probabilistic string similarity joins[C]// Proceedings of 29th ACM SIGMOD International Conference on Management of Data. New York: ACM, 2010: 327 - 338.

[3] BRYAN B, EBERHARDT F, FALOUTSOS C. Compact similarity joins [C]// ICDE 2008: Proceeding of the 24th International Conference on Data Engineering. Piscataway: IEEE, 2008: 346 - 355.

[4] XIAO C, WANG W, LIN X M, *et al*. Efficient similarity joins for near duplicate detection [C]// WWW'08: Proceedings of the 17th International Conference on World Wide Web. New York: ACM, 2011: 695 - 704.

[5] FENG J H, WANG J N, LI G L. Trie-Join: a Trie-based method for efficient string similarity joins [J]. The VLDB Journal, 2012, 21(4): 437 - 461.

[6] 李璐, 王宏志, 李建中, 等. Ed-Sjoin: 一种优化的字符串相似连接算法[J]. 计算机研究与发展, 2009, 46(Suppl.): 319 - 325.

[7] FENG J H, LI G L. Efficient fuzzy type-ahead search in XML data [J]. IEEE Transactions on knowledge and Data Engineering, 2012, 24(5): 882 - 895.

[8] FENG J H, LI G L, WANG J Y. Finding Top-*k* answers in keyword search over relational database using tuple units [J]. IEEE Transactions on Knowledge and Data Engineering, 2011, 23(12): 1781 - 1794.

[9] FENG J H, LI G L, WANG J Y, *et al*. Finding and ranking compact connected trees for effective keyword proximity search in XML documents[J]. Information Systems, 2010, 35(2): 186 - 203.

[10] AGRAWAL P, WIDOM J. Confidence-aware join algorithms [C]// ICDE 2009: Proceedings of the 25th International Conference on Data Engineering. Washington, DC: IEEE Computer Society, 2009: 628 - 639.

[11] ARASU A, GANTI V, KAUSHIK R. Efficient exact set-similarity joins [C]// VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases. [S. l.]: VLDB, 2006: 918 - 929.

[12] BAYARDO R J, MA Y, SRIKANT R. Scaling up all pairs similarity search [C]// WWW '07: Proceedings of the 16th International Conference on World Wide Web. New York: ACM, 2007: 131 - 140.

[13] NEUMANN T. Query simplification: graceful degradation for join-order optimization [C]// SIGMOD '09: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2009: 403 - 414.

[14] CHAUDHURI S, GANTI V, KAUSHIK R. A primitive operator for similarity joins in data cleaning [C]// ICDE '06: Proceedings of the 22nd International Conference on Data Engineering. Washington, DC: IEEE Computer Society, 2006: 5 - 16.

[15] CHAUDHURI S, KAUSHIK R. Extending autocompletion to tolerate errors [C]// SIGMOD '09: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2009: 707 - 718

Journal of Computer Applications 《计算机应用》